Back-Propagation

Lecture 0 2

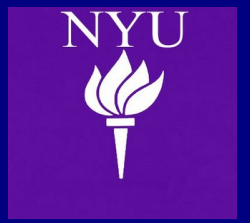
Yann Le Cun

Facebook AI Research,

Center for Data Science, NYU

Courant Institute of Mathematical Sciences, NYU

http://yann.lecun.com

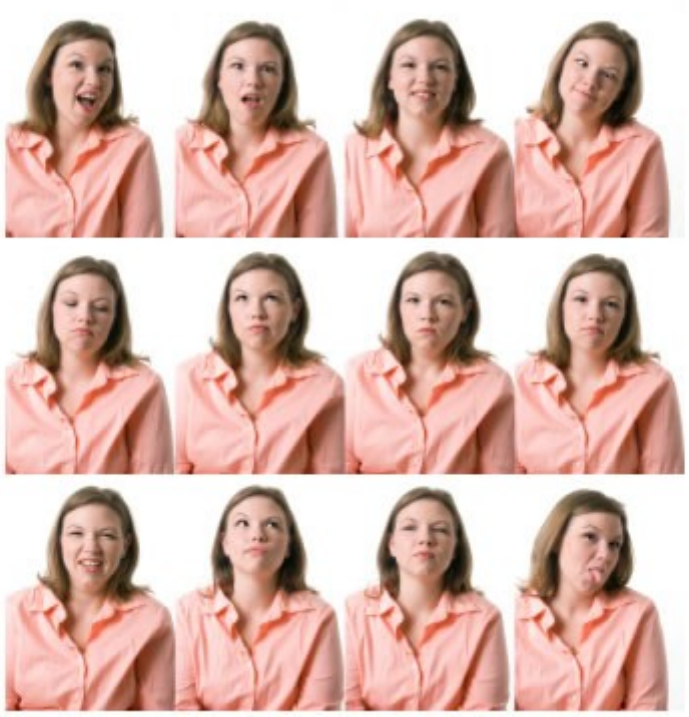


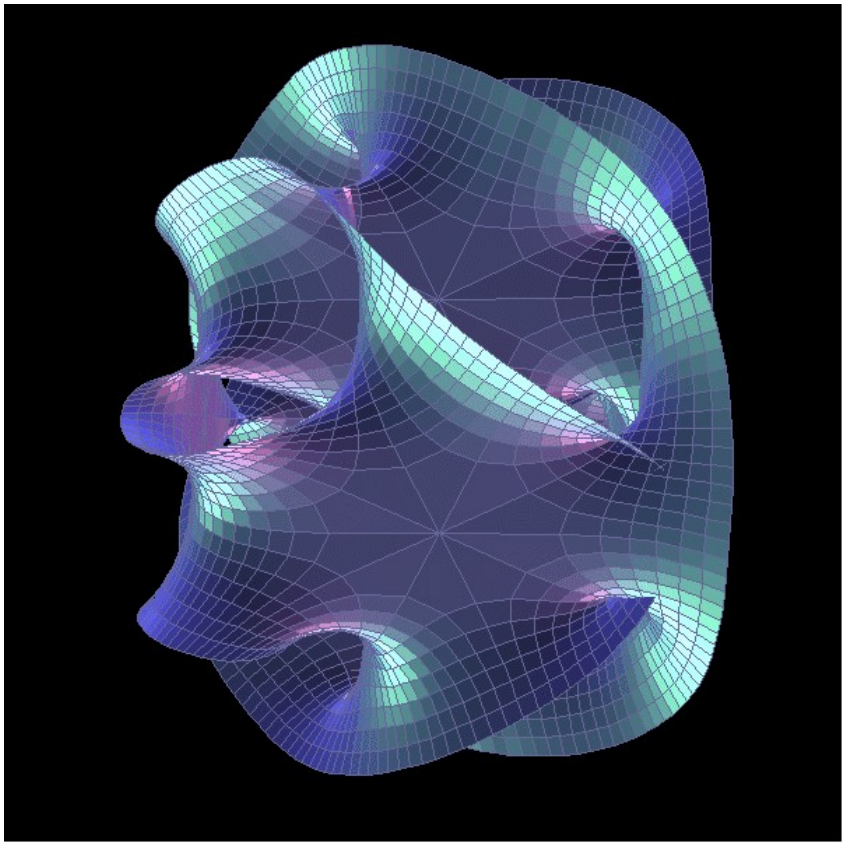

What Are
Good Feature?

Discovering the Hidden Structure in High-Dimensional Data
The manifold hypothesis

Y LeCun
MA Ranzato

- Learning Representations of Data:
  - **Discovering & disentangling the independent explanatory factors**
- The Manifold Hypothesis:
  - Natural data lives in a low-dimensional (non-linear) manifold
  - Because variables in natural data

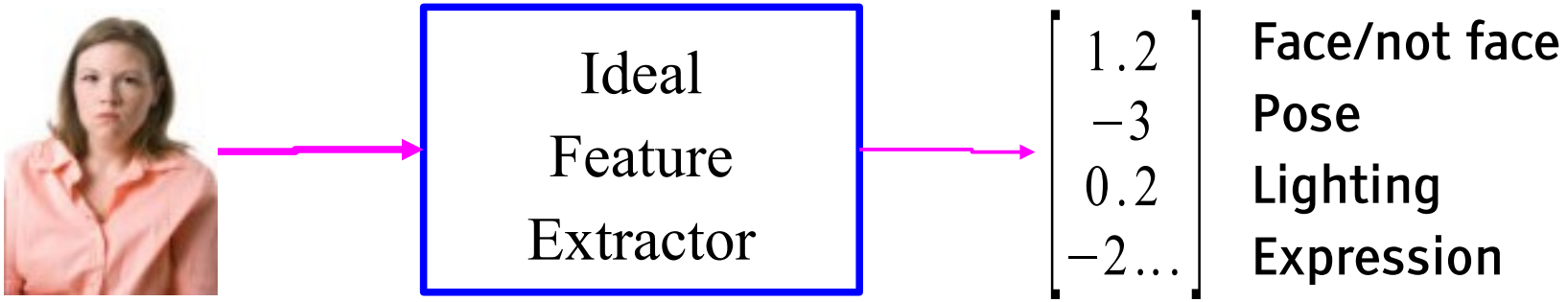




Discovering the Hidden Structure in High-Dimensional Data

Y LeCun
MA Ranzato

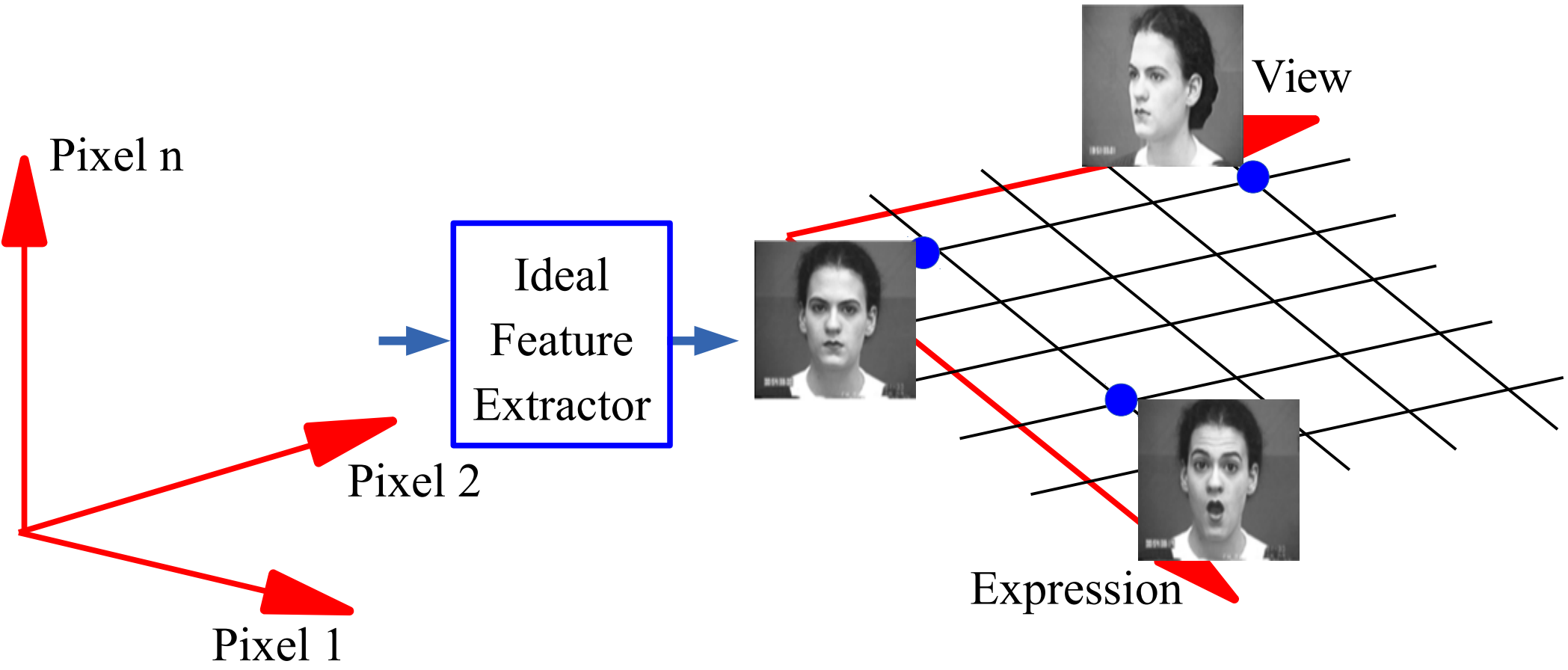
- **Example: all face images of a person**
  - 1000x1000 pixels = 1,000,000 dimensions
  - But the face has 3 cartesian coordinates and 3 Euler angles
  - And humans have less than about 50 muscles in the face
  - Hence the manifold of face images for a person has <56 dimensions
- **The perfect representations of a face image:**
  - Its coordinates on the face manifold
  - Its coordinates away from the manifold
- **We do not have good and general methods to learn functions that turns an image into this kind of representation**



Disentangling factors of variation

Y LeCun
MA Ranzato

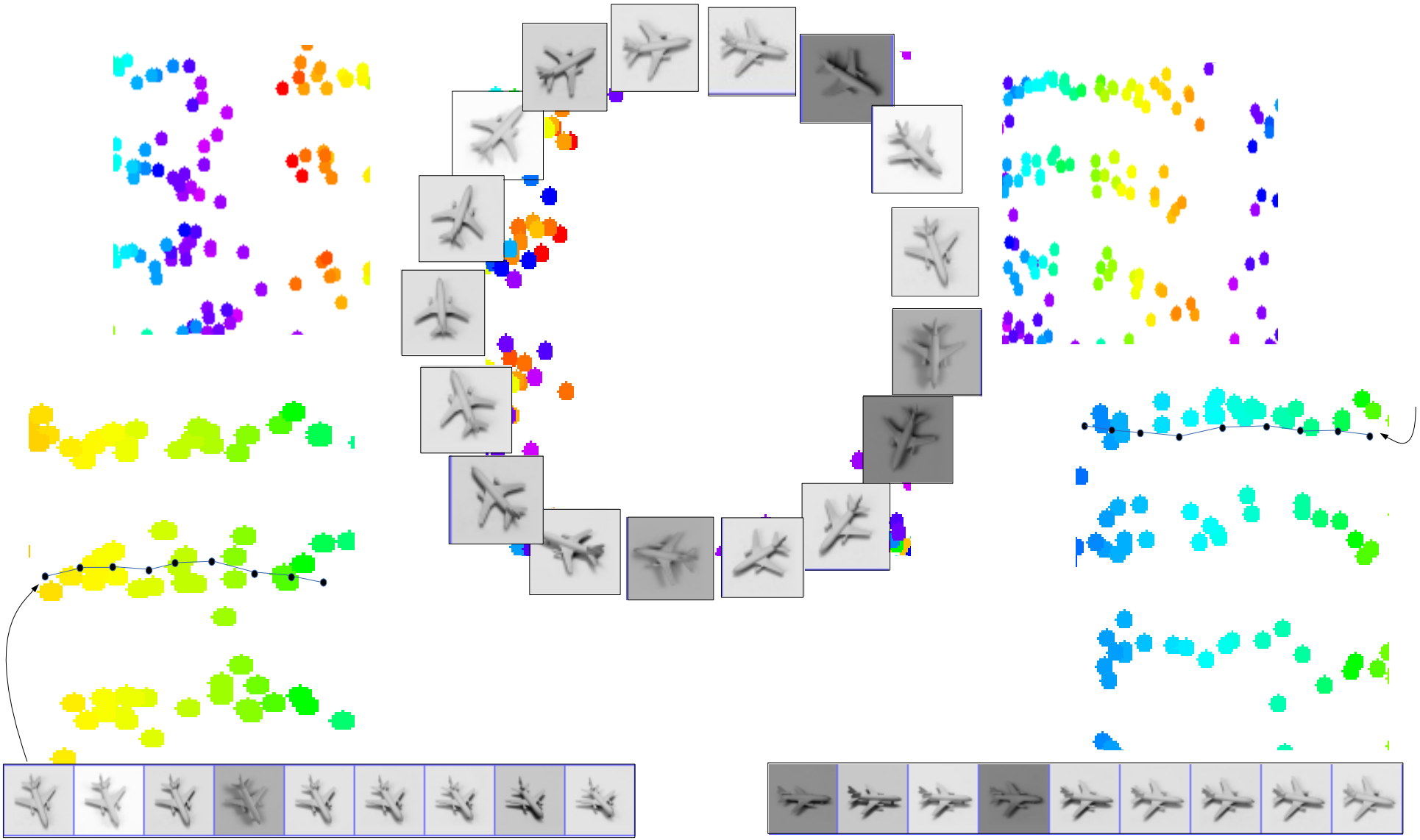
The Ideal Disentangling Feature Extractor



Data Manifold & Invariance: Some variations must be eliminated

Y LeCun
MA Ranzato

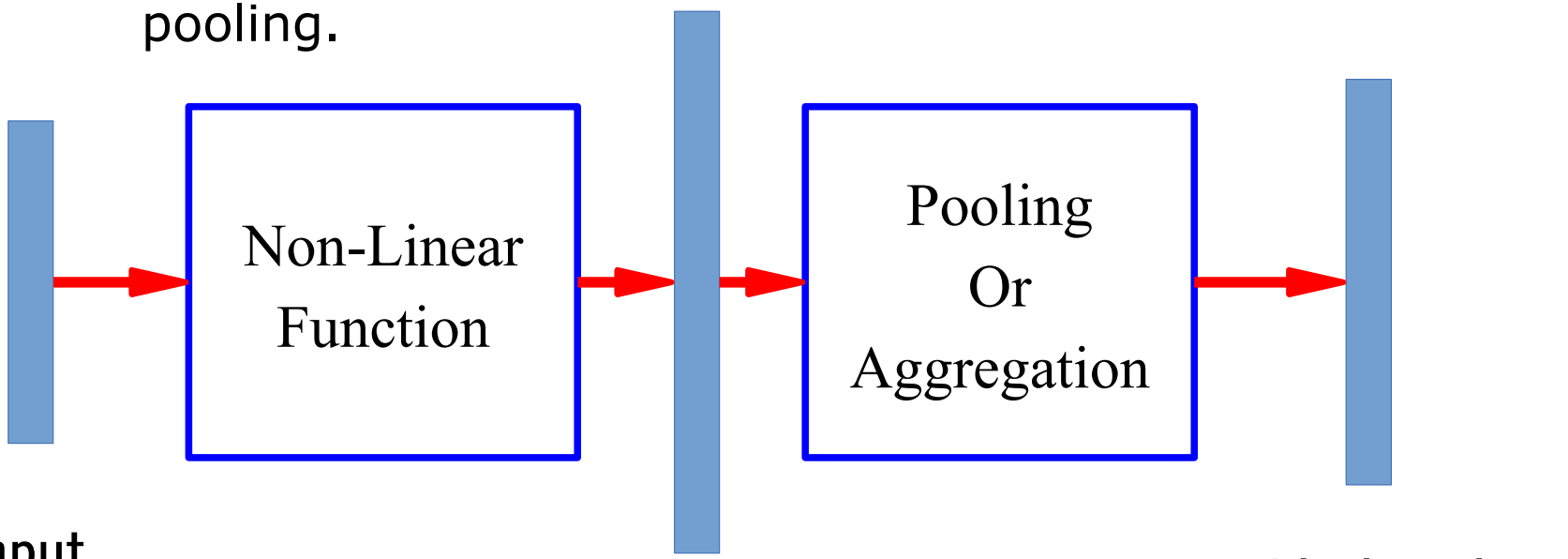
- Azimuth-Elevation manifold. Ignores lighting. [Hadsell et al. CVPR 2006]



Basic Idea for Invariant Feature Learning

Y LeCun
MA Ranzato

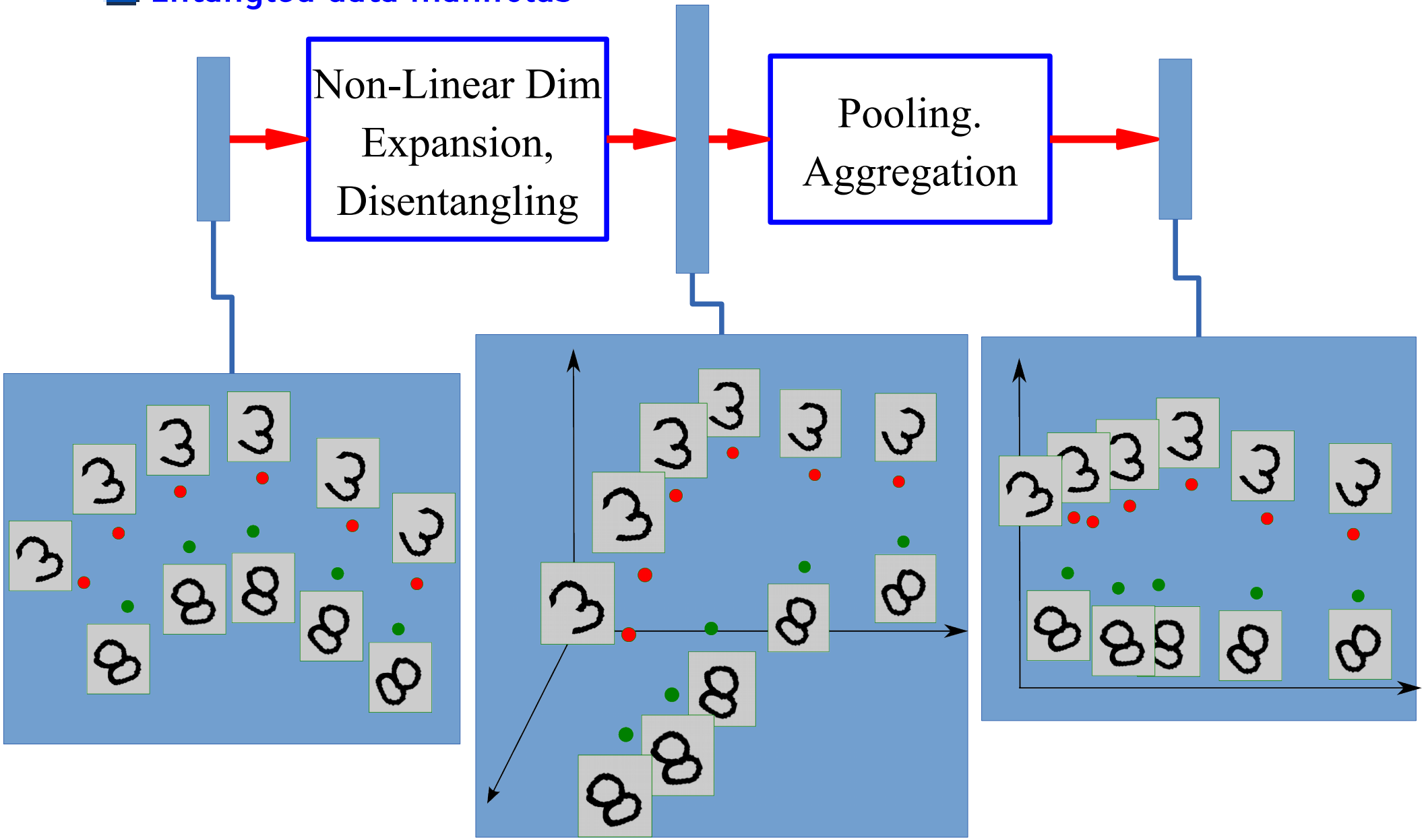
- Embed the input non-linearly into a high(er) dimensional space
  - In the new space, things that were non separable may become separable
- Pool regions of the new space together
  - Bringing together things that are semantically similar. Like pooling.



Non-Linear Expansion → Pooling

Y LeCun
MA Ranzato

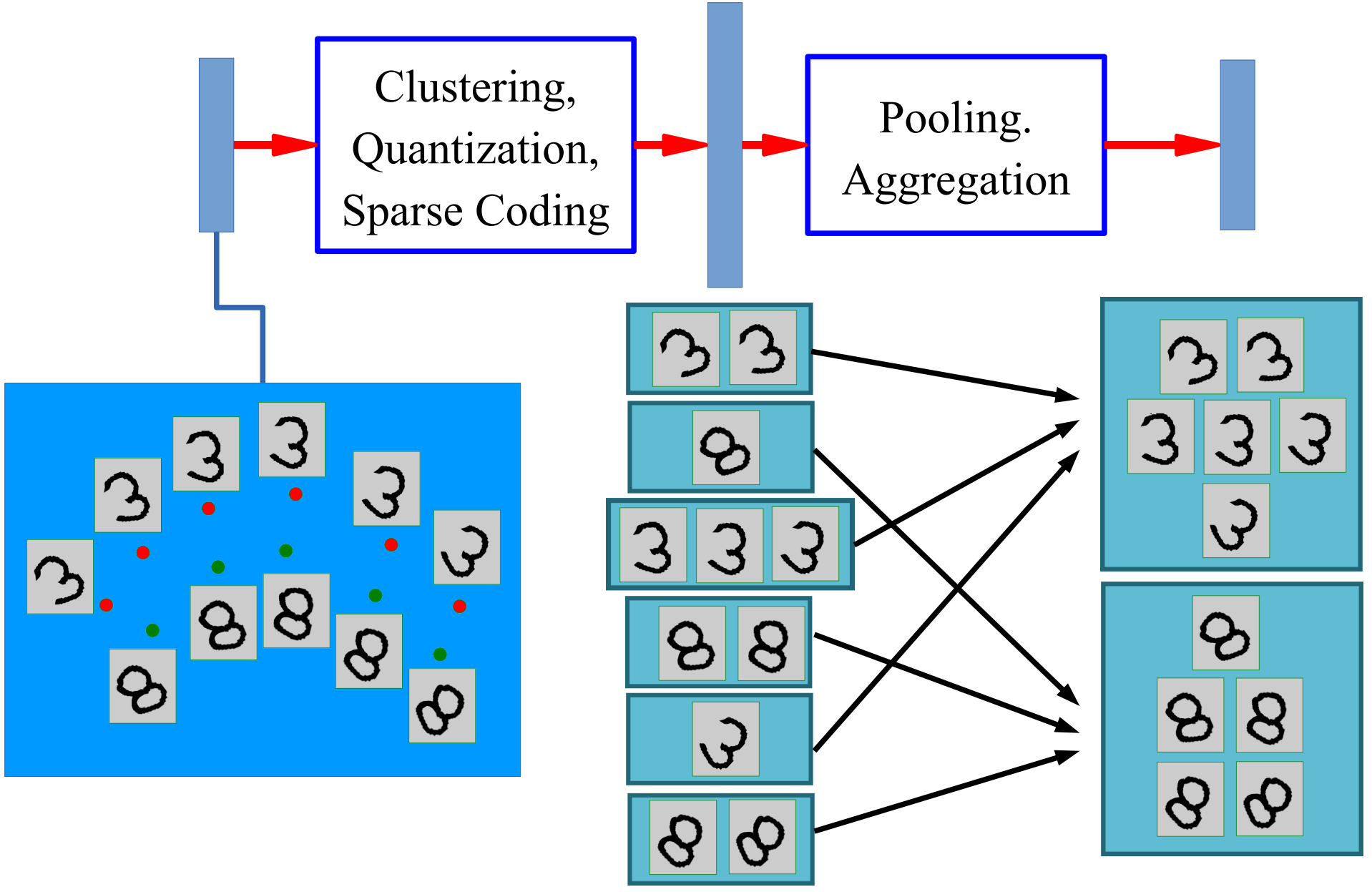
- Entangled data manifolds



Sparse Non-Linear Expansion → Pooling

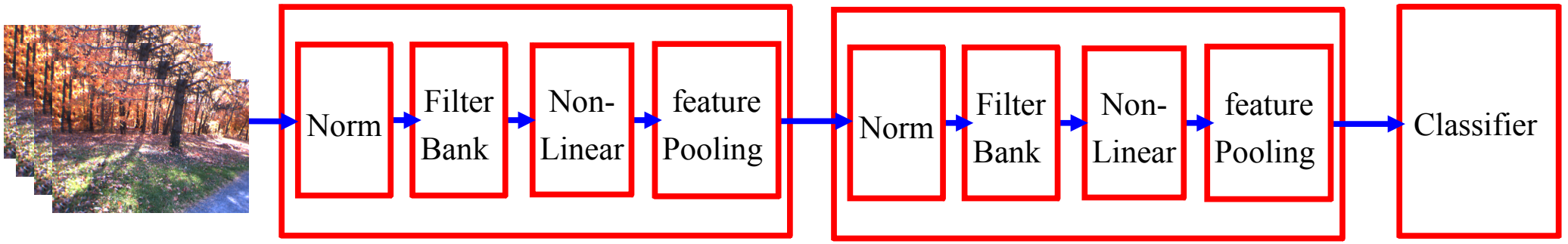
Y LeCun
MA Ranzato

- Use clustering to break things apart, pool together similar things



Overall Architecture: Normalization → Filter Bank → Non-Linearity → Pooling

Y LeCun
MA Ranzato

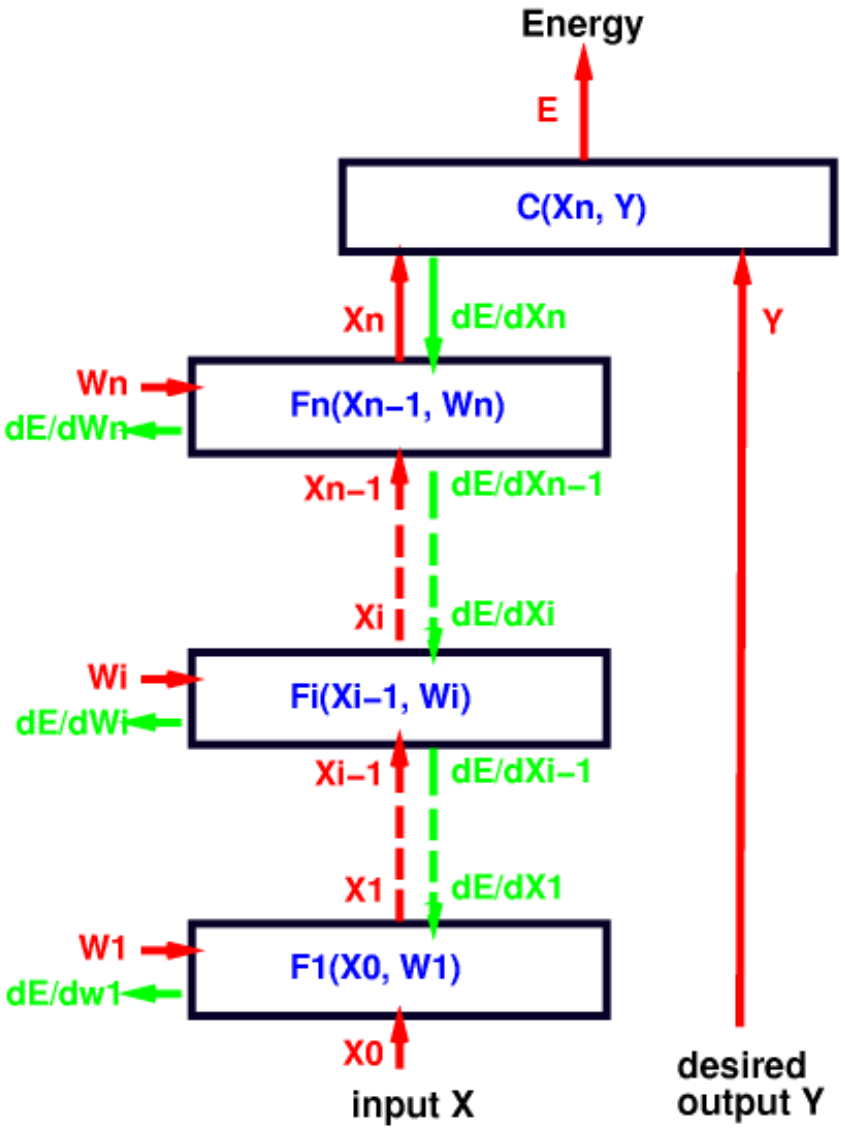


- Stacking multiple stages of
  - [Normalization → Filter Bank → Non-Linearity → Pooling].
- Normalization: variations on whitening
  - Subtractive: average removal, high pass filtering
  - Divisive: local contrast normalization, variance normalization
- Filter Bank: dimension expansion, projection on overcomplete basis
- Non-Linearity: sparsification, saturation, lateral inhibition....
  - Rectification (ReLU), Component-wise shrinkage, tanh, winner-takes-all
- Pooling: aggregation over space or feature type
  - $X_i; \quad L_p: \sqrt[p]{X_i^p}; \quad PROB: \frac{1}{b}\log\left(\sum_i e^{bX_i}\right)$


Deep Supervised Learning
(modular approach)

Multimodule Systems: Cascade

Y LeCun
MA Ranzato



- Complex learning machines can be built by assembling modules into networks
- Simple example: sequential/layered feed-forward architecture (cascade)
- Forward Propagation:

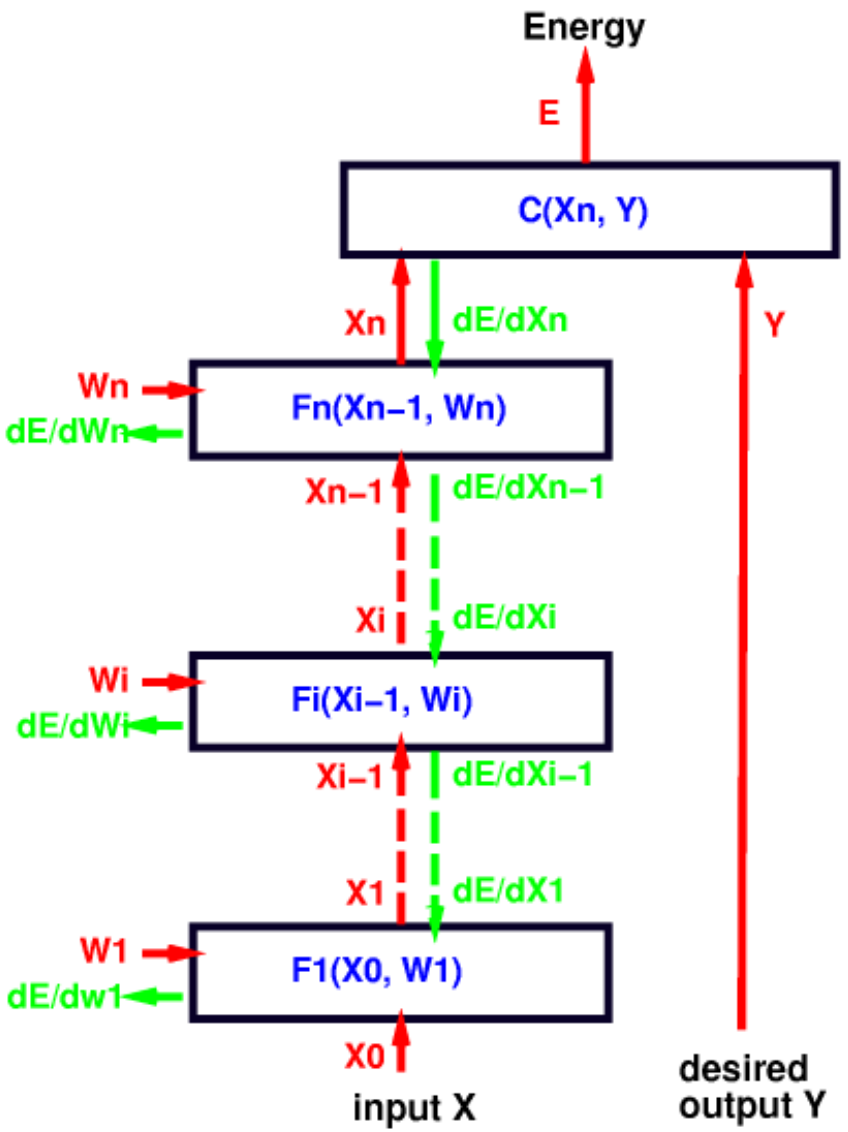
$$\text{let } X = X_0,$$

$$X_i = F_i(X_{i-1}, W_i) \quad \forall i \in [1, n]$$

$$E(Y, X, W) = C(X_n, Y)$$

Multimodule Systems: Implementation

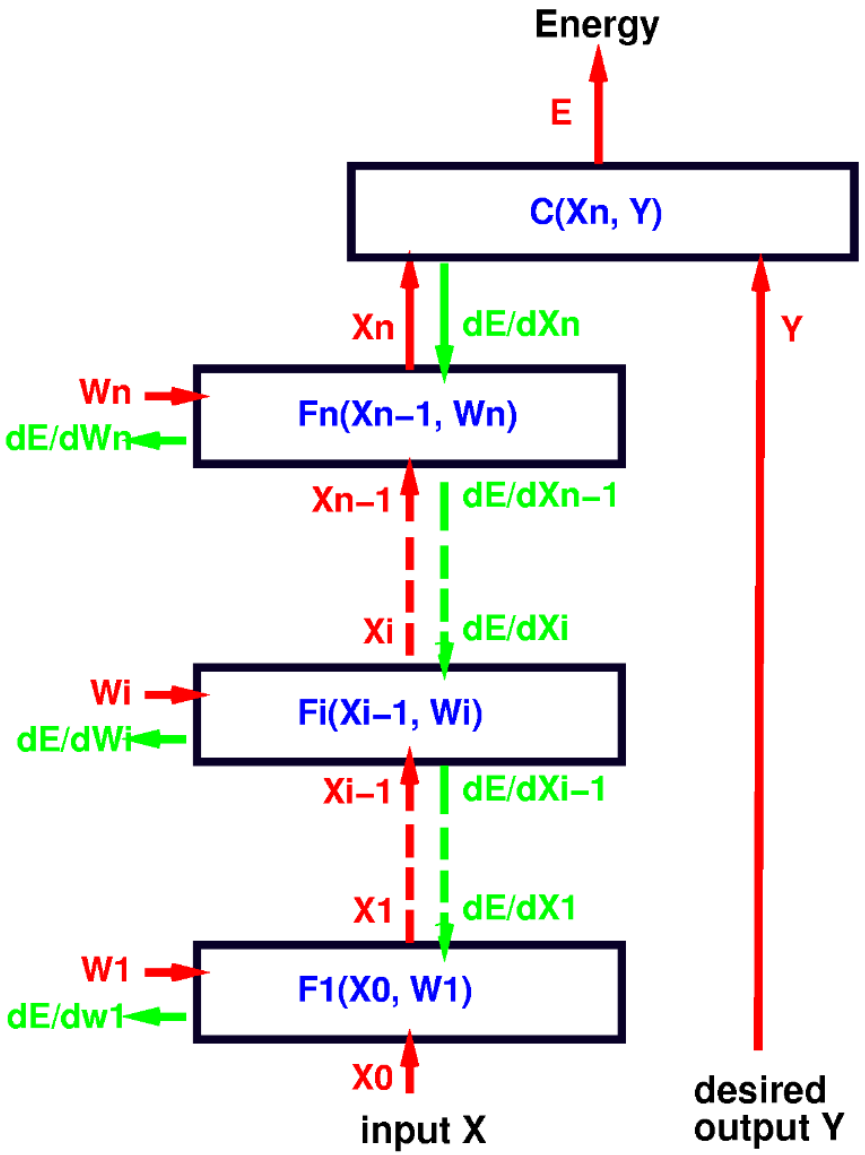
Y LeCun
MA Ranzato



- Each module is an object
  - Contains trainable parameters
  - Inputs are arguments
  - Output is returned, but also stored internally
  - Example: 2 modules m1, m2
- Torch7 (by hand)
  - `hid = m1:forward(in)`
  - `out = m2:forward(hid)`
- Torch7 (using the nn.Sequential class)
  - `model = nn.Sequential()`
  - `model:add(m1)`
  - `model:add(m2)`
  - `out = model:forward(in)`

Computing the Gradient in Multi-Layer Systems

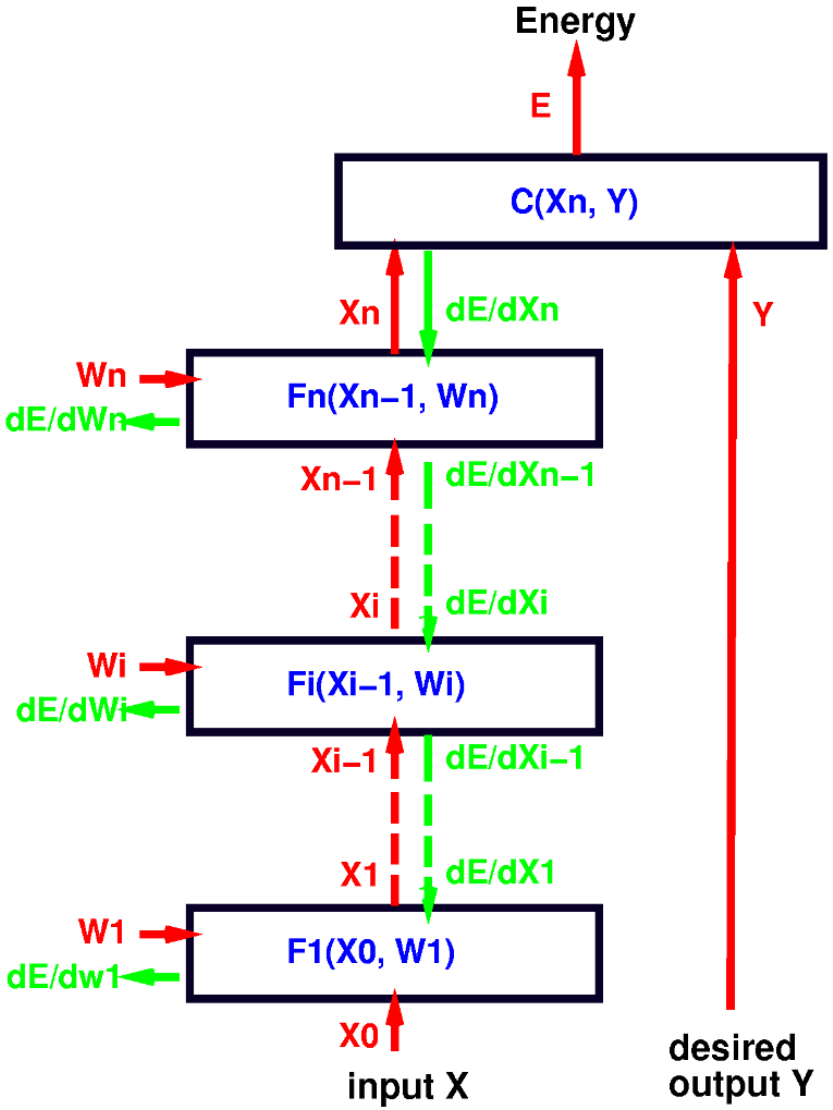
Y LeCun
MA Ranzato



- To train a multi-module system, we must compute the gradient of $E(W, Y, X)$ with respect to all the parameters in the system (all the W_k).
- Let's consider module i whose fprop method computes $X_k = F_k(X_{k-1}, W_k)$.
- Let's assume that we already know $\frac{\partial E}{\partial X_k}$, in other words, for each component of vector X_k we know how much E would wiggle if we wiggled that component of X_k.

Computing the Gradient in Multi-Layer Systems

Y LeCun
MA Ranzato



- We can apply chain rule to compute $\frac{\partial E}{\partial W_k}$ (how much E would wiggle if we wiggled each component of W_k):

$$\frac{\partial E}{\partial W_k} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial W_k}$$

$$[1 \times N_w] = [1 \times N_x].[N_x \times N_w]$$

- $\frac{\partial F_k(X_{k-1}, W_k)}{\partial W_k}$ is the *Jacobian matrix* of F_k with respect to W_k.

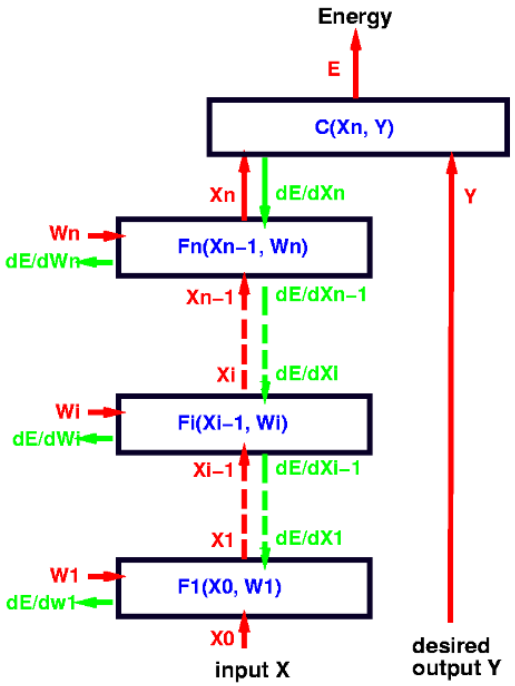
$$\left[\frac{\partial F_k(X_{k-1}, W_k)}{\partial W_k}\right]_{pq} = \frac{\partial \left[F_k(X_{k-1}, W_k)\right]_p}{\partial [W_k]_q}$$

- Element (p, q) of the Jacobian indicates how much the p-th output wiggles when we wiggle the q-th weight.

Computing the Gradient in Multi-Layer Systems

Y LeCun
MA Ranzato

Using the same trick, we can compute $\frac{\partial E}{\partial X_{k-1}}$. Let's assume again that we already know $\frac{\partial E}{\partial X_k}$, in other words, for each component of vector X_k we know how much E would wiggle if we wiggled that component of X_k.



- We can apply chain rule to compute $\frac{\partial E}{\partial X_{k-1}}$ (how much E would wiggle if we wiggled each component of X_{k-1}):

$$\frac{\partial E}{\partial X_{k-1}} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}$$

- $\frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}$ is the *Jacobian matrix* of F_k with respect to X_{k-1}.
- F_k has two Jacobian matrices, because it has to arguments.
- Element (p, q) of this Jacobian indicates how much the p-th output wiggles when we wiggle the q-th input.
- **The equation above is a recurrence equation!**

Jacobians and Dimensions

- derivatives with respect to a column vector are line vectors (dimensions: $[1 \times N_{k-1}] = [1 \times N_k] * [N_k \times N_{k-1}]$)

$$\frac{\partial E}{\partial X_{k-1}} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}$$

- (dimensions: $[1 \times N_{wk}] = [1 \times N_k] * [N_k \times N_{wk}]$):

$$\frac{\partial E}{\partial W_k} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial W}$$

- we may prefer to write those equation with column vectors:

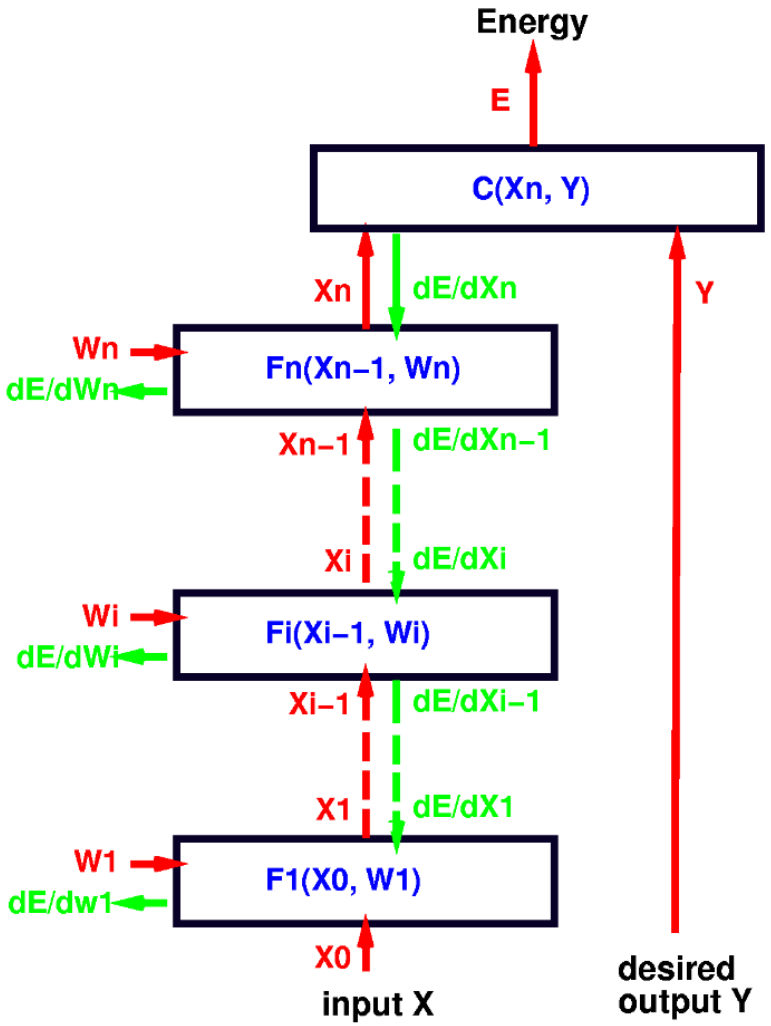
$$\frac{\partial E}{\partial X_{k-1}}' = \frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}' \frac{\partial E}{\partial X_k}'$$

$$\frac{\partial E}{\partial W_k}' = \frac{\partial F_k(X_{k-1}, W_k)}{\partial W}' \frac{\partial E}{\partial X_k}'$$

Back Propgation

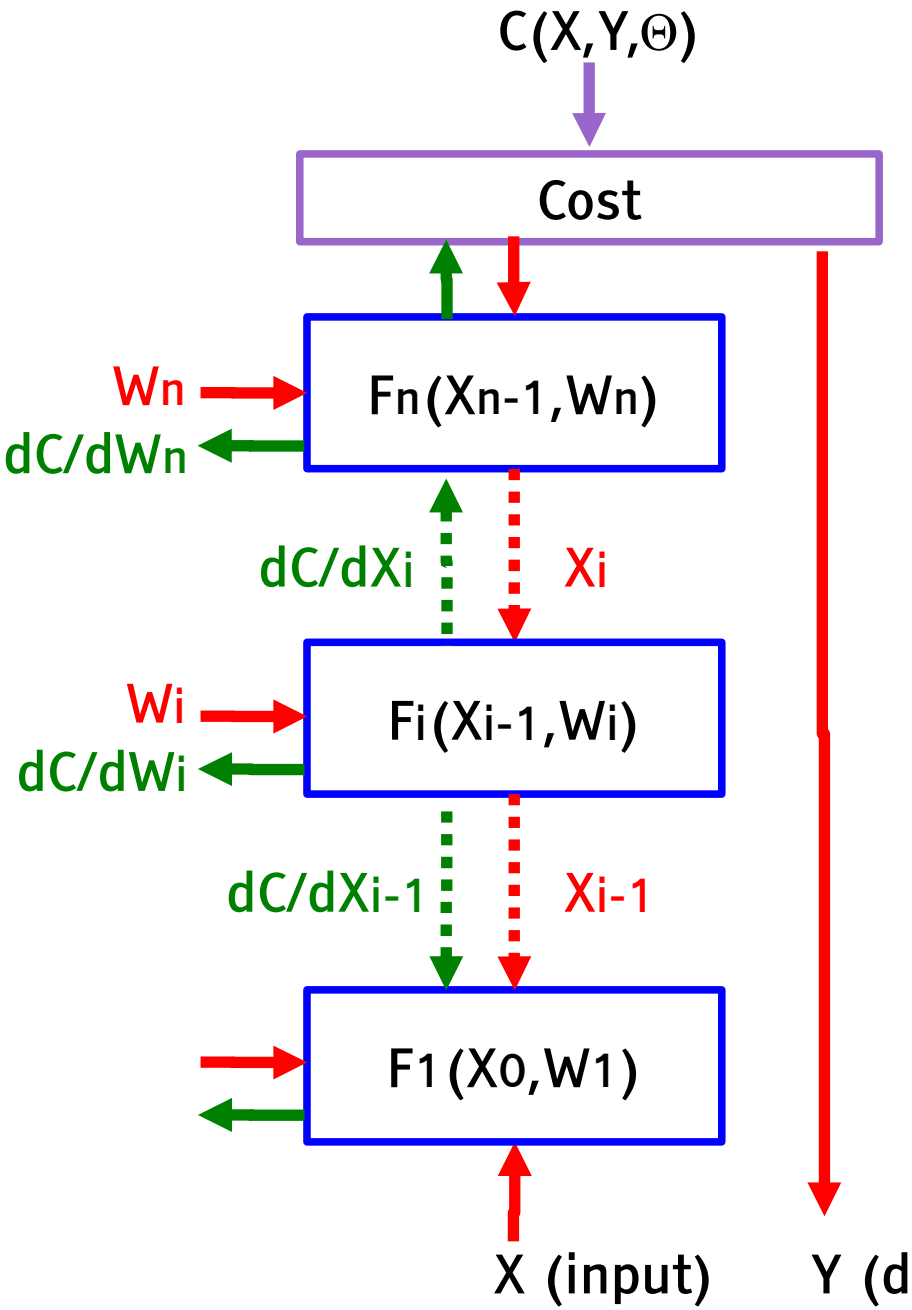
Y LeCun
MA Ranzato

To compute all the derivatives, we use a backward sweep called the **back-propagation algorithm** that uses the recurrence equation for $\frac{\partial E}{\partial X_k}$



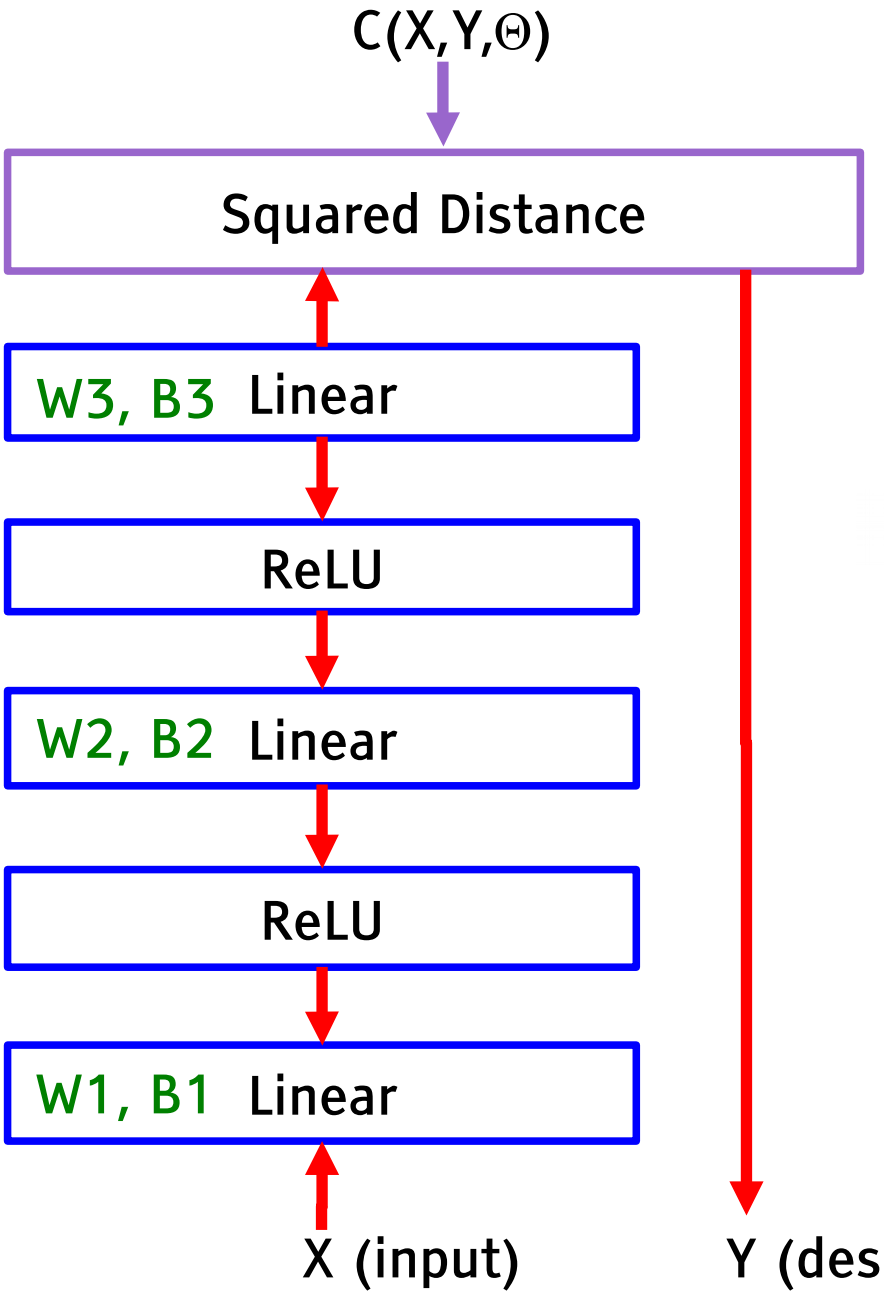
- $\frac{\partial E}{\partial X_n} = \frac{\partial C(X_n, Y)}{\partial X_n}$
- $\frac{\partial E}{\partial X_{n-1}} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(X_{n-1}, W_n)}{\partial X_{n-1}}$
- $\frac{\partial E}{\partial W_n} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(X_{n-1}, W_n)}{\partial W_n}$
- $\frac{\partial E}{\partial X_{n-2}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2}, W_{n-1})}{\partial X_{n-2}}$
- $\frac{\partial E}{\partial W_{n-1}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2}, W_{n-1})}{\partial W_{n-1}}$
- ....etc, until we reach the first module.
- we now have all the $\frac{\partial E}{\partial W_k}$ for $k \in [1, n]$.

Computing Gradients by Back-Propagation



- A practical Application of Chain Rule
- Backprop for the state gradients:
- dC/dXi-1 = dC/dXi . dXi/dXi-1
- dC/dXi-1 = dC/dXi . dFi(Xi-1,Wi)/dXi-1
- Backprop for the weight gradients:
- dC/dWi = dC/dXi . dXi/dWi
- dC/dWi = dC/dXi . dFi(Xi-1,Wi)/dWi

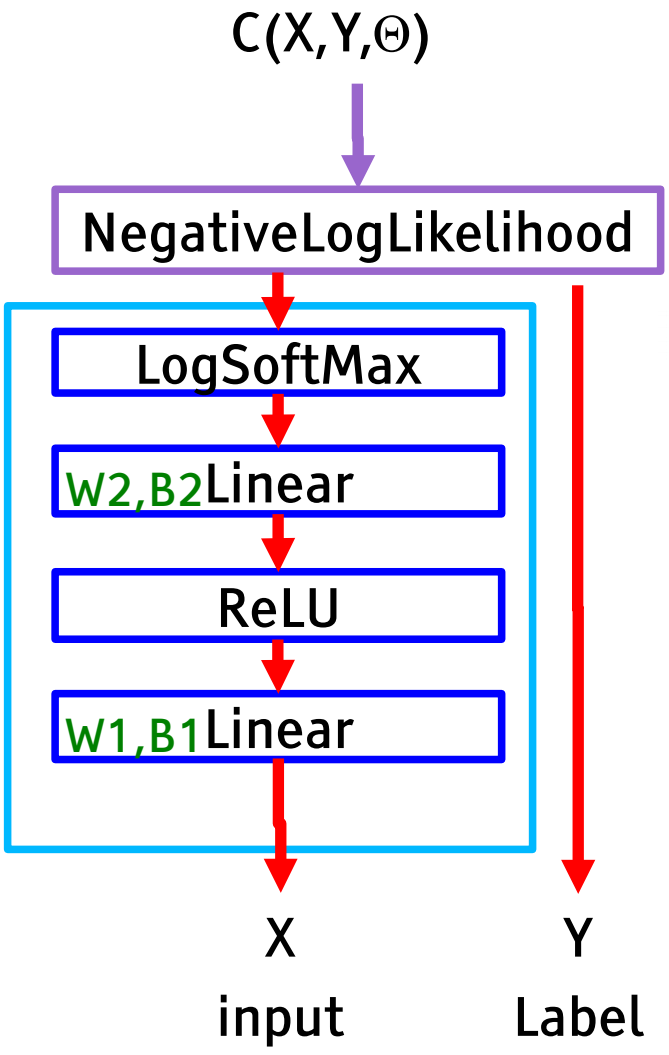
Typical Multilayer Neural Net Architecture



- Complex learning machines can be built by assembling modules into networks
- Linear Module
  - Out = W.In+B
- ReLU Module (Rectified Linear Unit)
  - $Out_i = 0$ if $In_i<0$
  - $Out_i = In_i$ otherwise
- Cost Module: Squared Distance
  - $C = ||In1 - In2||^2$
- Objective Function
  - $L(\Theta)=1/p \sum_k C(X^k,Y^k,\Theta)$
  - $\Theta = (W1,B1,W2,B2,W3,B3)$

Building a Network by Assembling Modules

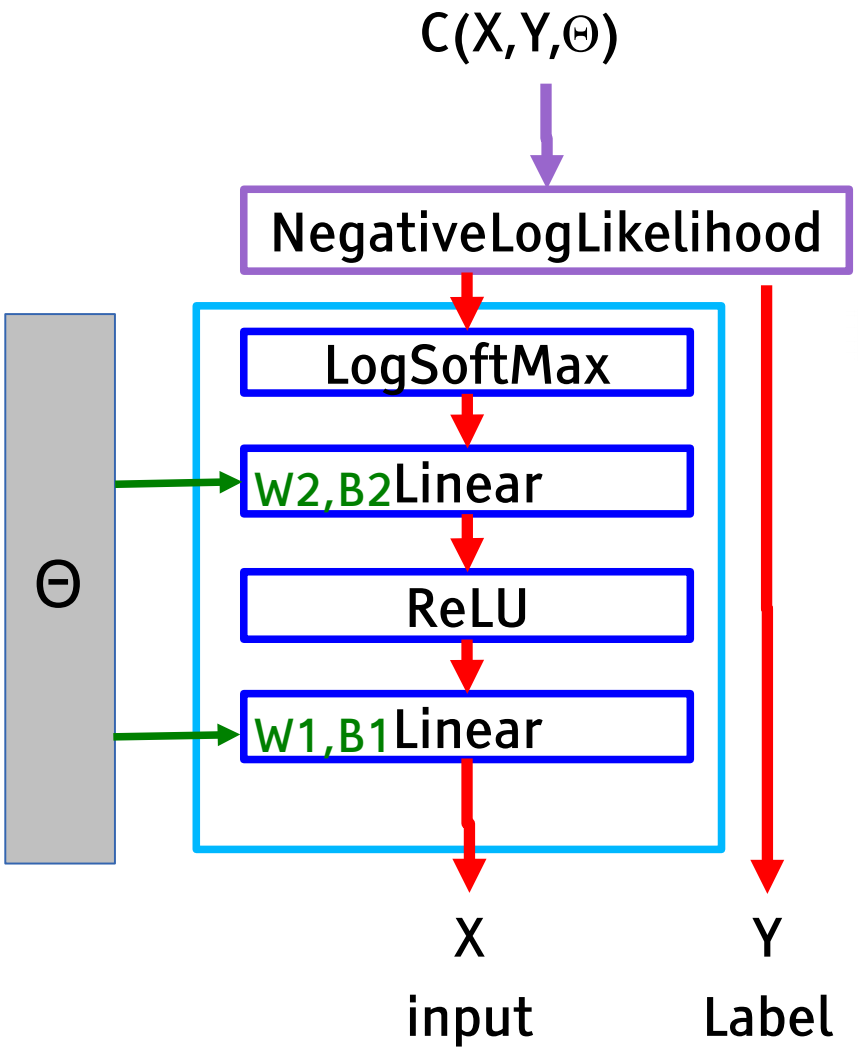
- All major deep learning frameworks use modules (inspired by SN/Lush, 1991)
  - Torch7, Theano, TensorFlow....




```
-- sizes
ninput = 28*28  -- e.g. for MNIST
nhidden1 = 1000
noutput = 10

-- network module
net = nn.Sequential()
net:add(nn.Linear(ninput, nhidden))
net:add(nn.Threshold())
net:add(nn.Linear(nhidden, noutput))
net:add(nn.LogSoftMax()))

-- cost module
cost = nn.ClassNLLCriterion()

-- get a training sample
input = trainingset.data[k]
target = trainingset.labels[k]

-- run through the model
output = net:forward(input)
c = cost:forward(output, target)
```

Running Backprop

- Torch7 example
- Gradtheta contains the gradient




```
-- network module
net = nn.Sequential()
net:add(nn.Linear(ninput, nhidden))
net:add(nn.Threshold())
net:add(nn.Linear(nhidden, noutput))
net:add(nn.LogSoftMax()))

-- cost module
cost = nn.ClassNLLCriterion()

-- gather the parameters in a vector
theta, gradtheta = net:getParameters()

-- get a training sample
input = trainingset.data[k]
target = trainingset.labels[k]

-- run through the model
output = net:forward(input)
c = cost:forward(output, target)

-- run backprop
gradtheta:zero()
gradoutput = cost:backward(output, target)
net:backward(input, gradoutput)
```

Module Classes

Linear • $Y = W.X$; $dC/dX = W^T.\, dC/dY$; $dC/dW = dC/dY\, .\, X^T$

ReLU • $y = ReLU(x)$; if $(x<0)$ $dC/dx = 0$ else $dC/dx = dC/dy$

Duplicate • $Y1 = X,\ Y2 = X$; $dC/dX = dC/dY1 + dC/dY2$

Add • $Y = X1 + X2$; $dC/dX1 = dC/dY$; $dC/dX2 = dC/dY$

Max • $y = max(x1,x2)$; if $(x1>x2)$ $dC/dx1 = dC/dy$ else $dC/dx1=0$

LogSoftMax • $Yi = Xi - \log\left[\sum_j \exp(Xj)\right]$; .....

Module Classes

Linear • $Y = W.X$; $dC/dX = W^T.\, dC/dY$; $dC/dW = dC/dY\, .\, X^T$

ReLU • $y = ReLU(x)$; if $(x<0)$ $dC/dx = 0$ else $dC/dx = dC/dy$

Duplicate • $Y1 = X,\ Y2 = X$; $dC/dX = dC/dY1 + dC/dY2$

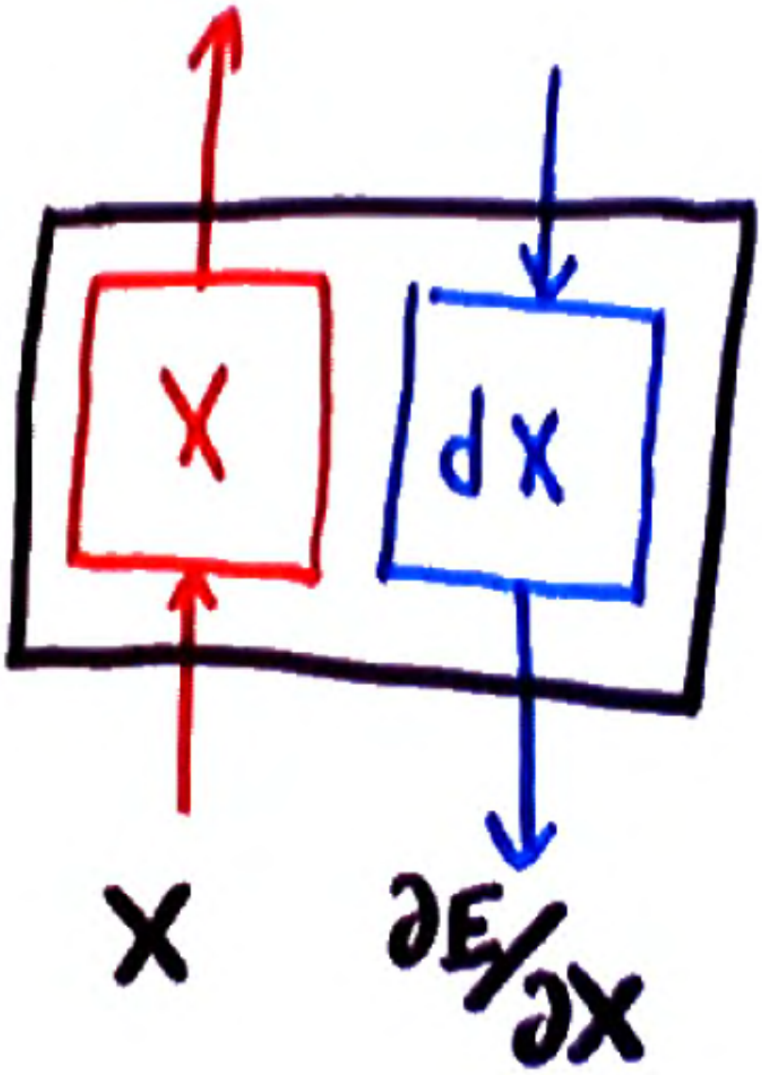
Add • $Y = X1 + X2$; $dC/dX1 = dC/dY$; $dC/dX2 = dC/dY$

Max • $y = max(x1,x2)$; if $(x1>x2)$ $dC/dx1 = dC/dy$ else $dC/dx1=0$

LogSoftMax • $Yi = Xi - \log\left[\sum_j \exp(Xj)\right]$; .....

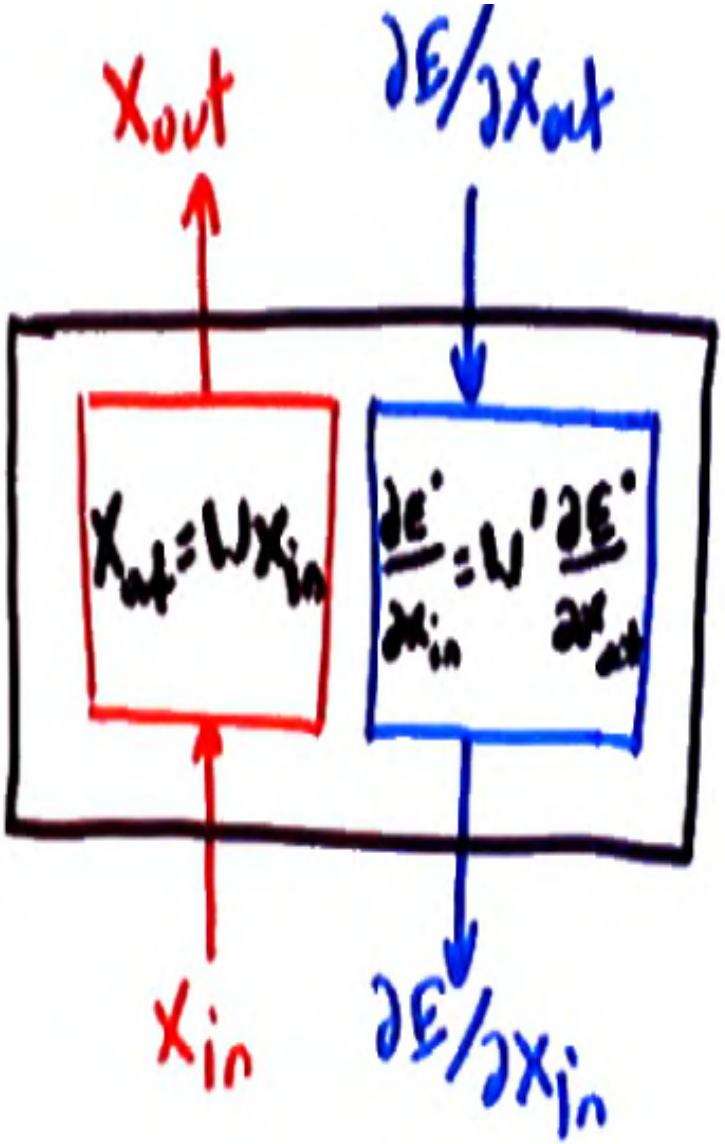
State Variables

Y LeCun
MA Ranzato



the internal state of the network will be kept in a "state" class that contains two scalars, vectors, or matrices: (1) the state proper, (2) the derivative of the energy with respect to that state.

Linear Module



- fprop: $X_{\text{out}} = W X_{\text{in}}$
- bprop to input:

$$\frac{\partial E}{\partial X_{\text{in}}} = \frac{\partial E}{\partial X_{\text{out}}} \frac{\partial X_{\text{out}}}{\partial X_{\text{in}}} = \frac{\partial E}{\partial X_{\text{out}}} W$$

- by transposing, we get column vectors:

$$\frac{\partial E}{\partial X_{\text{in}}}' = W' \frac{\partial E}{\partial X_{\text{out}}}'$$

- bprop to weights:

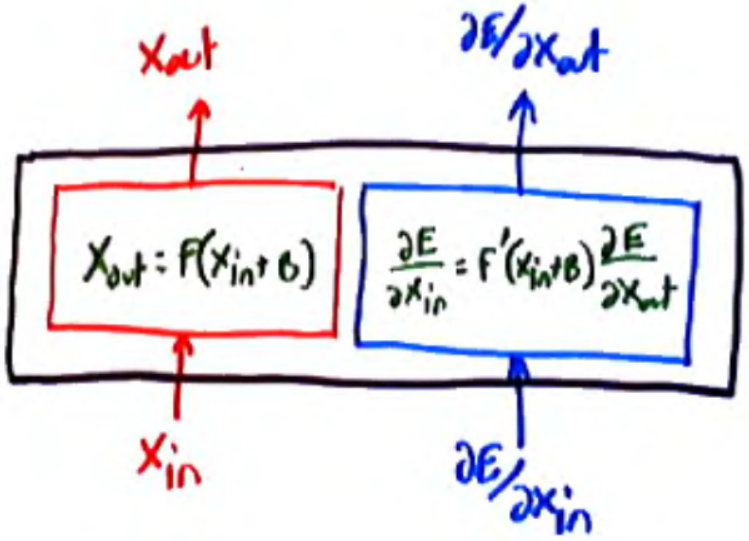
$$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial X_{\text{out}i}} \frac{\partial X_{\text{out}i}}{\partial W_{ij}} = X_{\text{in}j} \frac{\partial E}{\partial X_{\text{out}i}}$$

- We can write this as an outer-product:

$$\frac{\partial E}{\partial W}' = \frac{\partial E}{\partial X_{\text{out}}}' X_{in}'$$

Tanh module (or any other pointwise function)

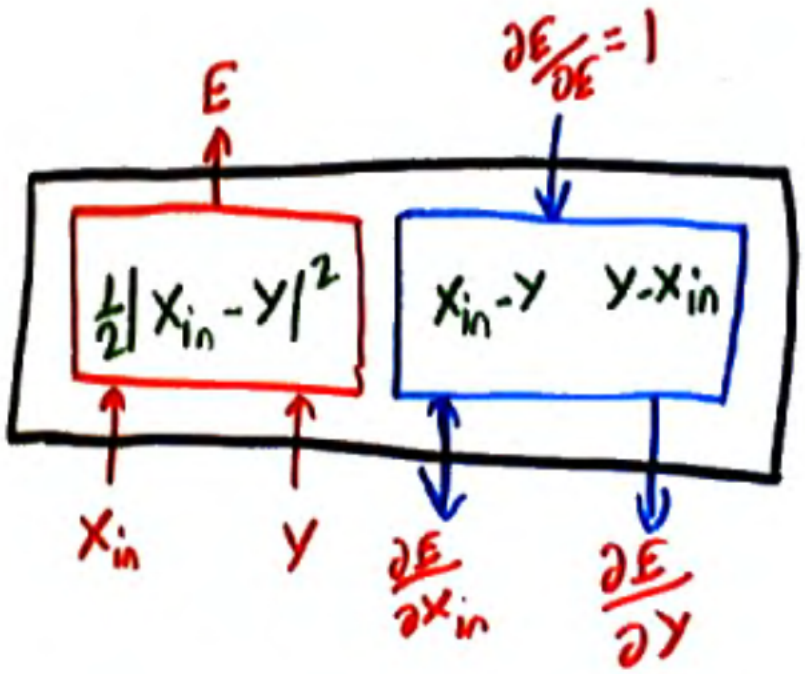
Y LeCun
MA Ranzato



- fprop: $(X_{\text{out}})_i = \tanh((X_{\text{in}})_i + B_i)$
- bprop to input:
  $(\frac{\partial E}{\partial X_{\text{in}}})_i = (\frac{\partial E}{\partial X_{\text{out}}})_i \tanh'((X_{\text{in}})_i + B_i)$
- bprop to bias:
  $\frac{\partial E}{\partial B_i} = (\frac{\partial E}{\partial X_{\text{out}}})_i \tanh'((X_{\text{in}})_i + B_i)$
- $\tanh(x) = \frac{2}{1+\exp -x} - 1 = \frac{1-\exp(-x)}{1+\exp(-x)}$

Euclidean Distance Module (Squared Error)

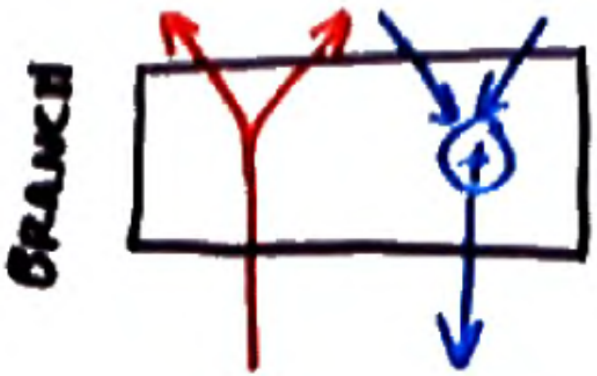
Y LeCun
MA Ranzato



- fprop: $X_{\text{out}} = \frac{1}{2}||X_{\text{in}} - Y||^2$
- bprop to X input: $\frac{\partial E}{\partial X_{\text{in}}} = X_{\text{in}} - Y$
- bprop to Y input: $\frac{\partial E}{\partial Y} = Y - X_{\text{in}}$

Y connector and Addition modules

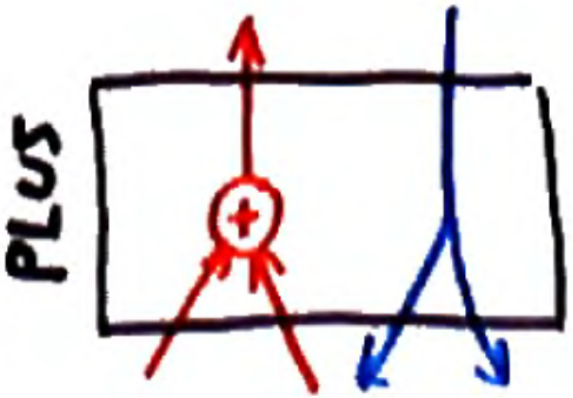
Y LeCun
MA Ranzato



- The PLUS module: a module with K inputs $X_1, \ldots, X_K$ (of any type) that computes the sum of its inputs:

$$X_{\text{out}} = \sum_k X_k$$

back-prop: $\frac{\partial E}{\partial X_k} = \frac{\partial E}{\partial X_{\text{out}}} \quad \forall k$

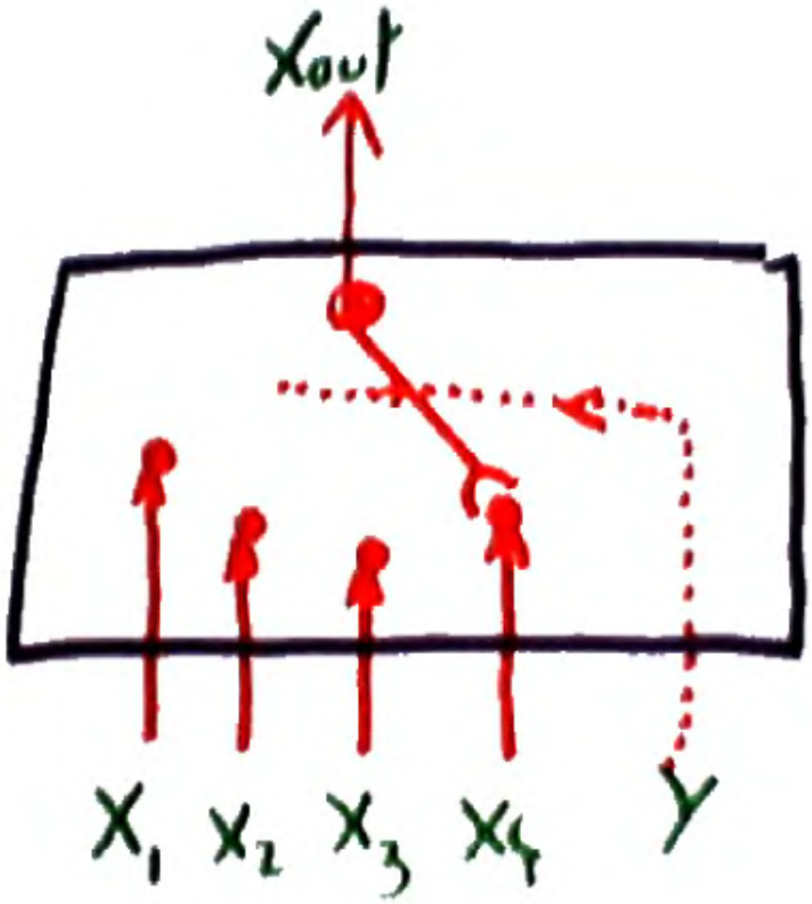


- The BRANCH module: a module with one input and K outputs $X_1, \ldots, X_K$ (of any type) that simply copies its input on its outputs:

$$X_k = X_{\text{in}} \quad \forall k \in [1..K]$$

back-prop: $\frac{\partial E}{\partial \text{in}} = \sum_k \frac{\partial E}{\partial X_k}$

Switch Module



- A module with K inputs $X_1, \ldots, X_K$ (of any type) and one additional discrete-valued input Y.
- The value of the discrete input determines which of the N inputs is copied to the output.
-

$$X_{\text{out}} = \sum_k \delta(Y-k) X_k$$

$$\frac{\partial E}{\partial X_k} = \delta(Y-k) \frac{\partial E}{\partial X_{\text{out}}}$$

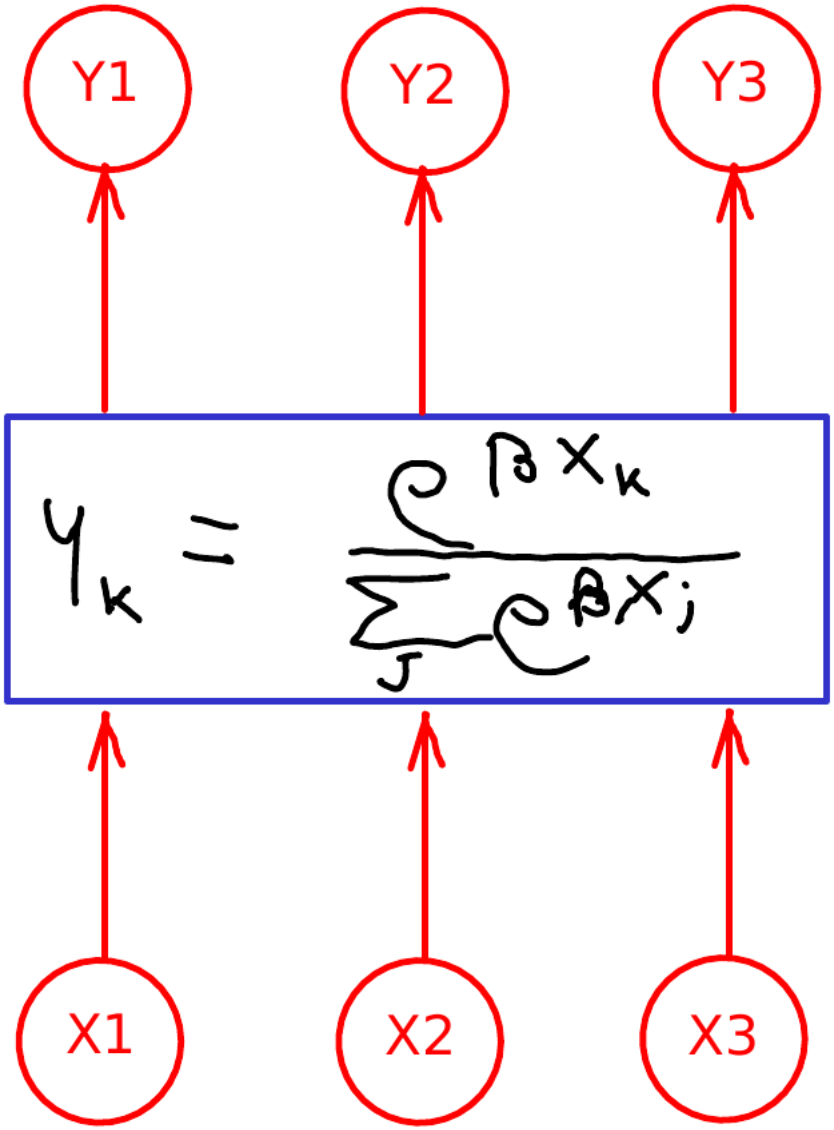
the gradient with respect to the output is copied to the gradient with respect to the switched-in input. The gradients of all other inputs are zero.

SoftMax Module (should really be called SoftArgMax)

Y LeCun
MA Ranzato

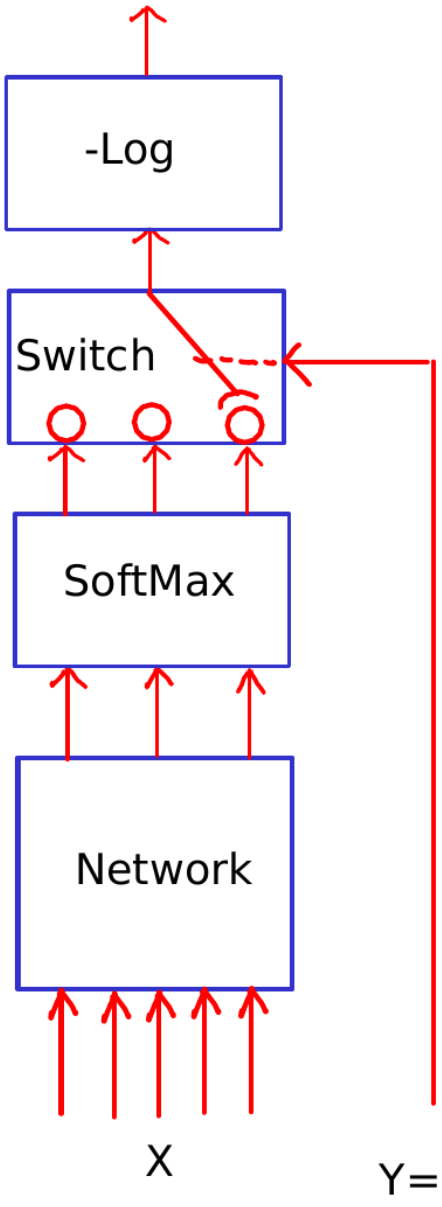
- Transforms scores into a discrete probability distribution
  - Positive numbers that sum to one.
- Used in multi-class classification

$$P_k = \frac{e^{\beta X_k}}{\sum_j e^{\beta X_j}}$$

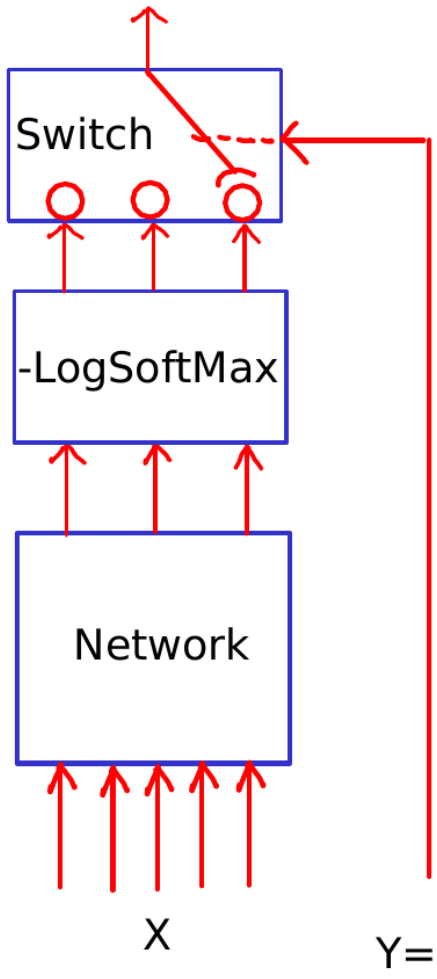


SoftMax Module: Loss Function for Classification

Y LeCun
MA Ranzato

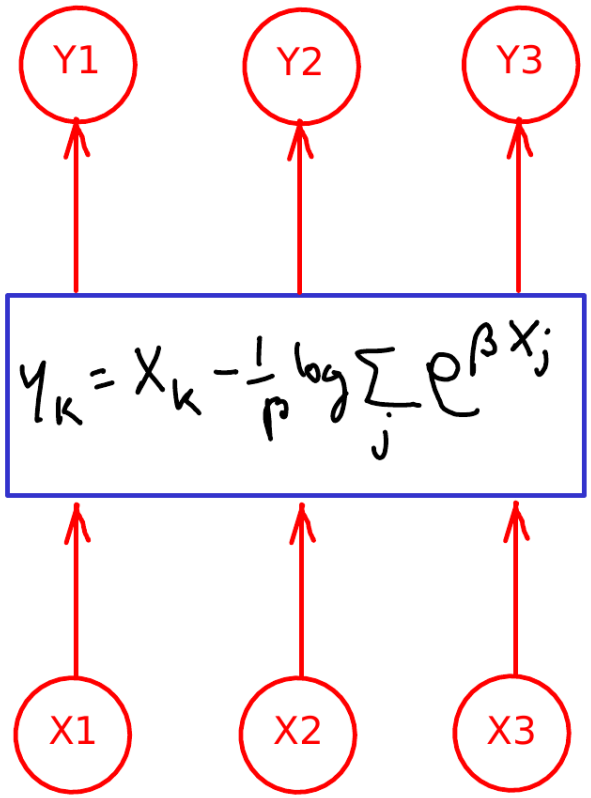


- -LogSoftMax: $-\frac{1}{\beta}\log p_h = -x_h + \frac{1}{\beta}\log\sum_j e^{\beta x_j}$
- Maximum conditional likelihood
- Minimize -log of the probability of the correct class.

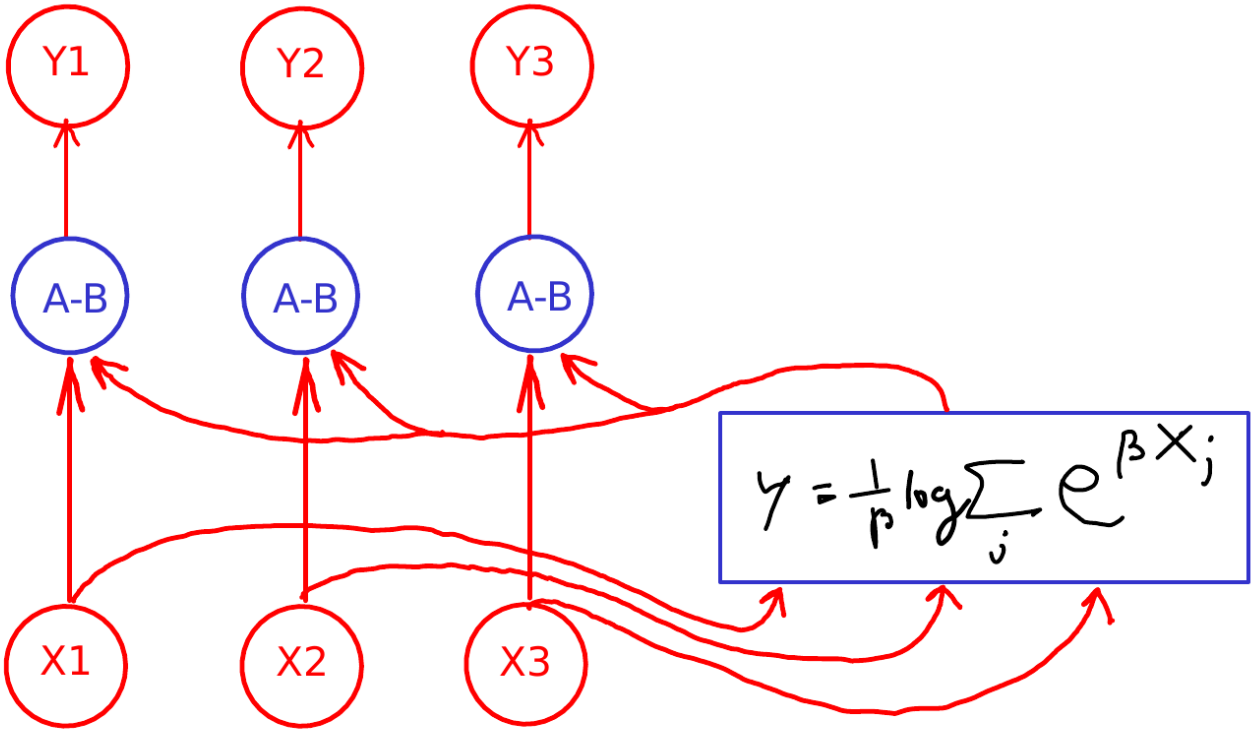


LogSoftMax Module

Y LeCun
MA Ranzato



- Transforms scores into a discrete probability distribution
- LogSoftMax = Identity - LogSumExp



LogSumExp Module

Y LeCun
MA Ranzato

- Log of normalization term for SoftMax
- Fprop

$$X_{out} = \frac{1}{\beta} \sum_j e^{\beta X_j}$$

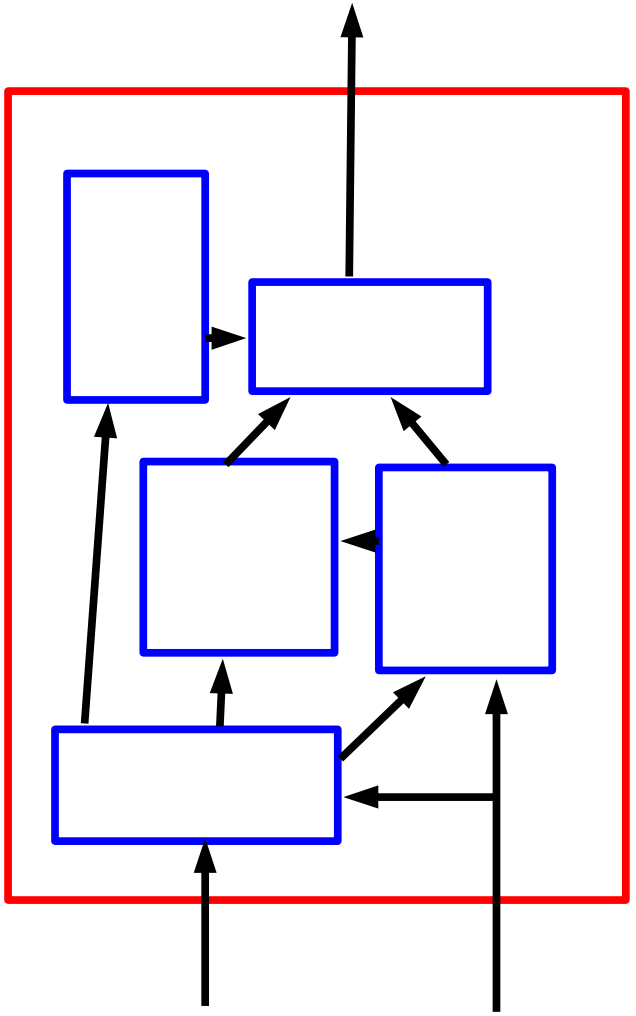
- Bprop

$$\frac{\partial E}{\partial X_k} = \frac{\partial E}{\partial X_{out}} \cdot \frac{e^{\beta X_k}}{\sum_j e^{\beta X_j}}$$

- Or:

$$\frac{\partial E}{\partial X_k} = \frac{\partial E}{\partial X_{out}} \cdot P_k \qquad P_k = \frac{e^{\beta X_k}}{\sum_j e^{\beta X_j}}$$

Backprop works through any modular architecture



- **Any connection is permissible**
  - Networks with loops must be "unfolded in time".
- **Any module is permissible**
  - As long as it is continuous and differentiable almost everywhere with respect to the parameters, and with respect to non-terminal inputs.

Backprop in Practice

- Use ReLU non-linearities (tanh and logistic are falling out of favor)
- Use cross-entropy loss for classification
- Use Stochastic Gradient Descent on minibatches
- Shuffle the training samples
- Normalize the input variables (zero mean, unit variance)
- Schedule to decrease the learning rate
- Use a bit of L1 or L2 regularization on the weights (or a combination)
  - But it's best to turn it on after a couple of epochs
- Use "dropout" for regularization
  - Hinton et al 2012 http://arxiv.org/abs/1207.0580
- Lots more in [LeCun et al. "Efficient Backprop" 1998]
- Lots, lots more in "Neural Networks, Tricks of the Trade" (2012 edition) edited by G. Montavon, G. B. Orr, and K-R Müller (Springer)

Example: building a Neural Net in Torch7

Y LeCun
MA Ranzato

- Net for SVHN digit recognition
- 10 categories
- Input is 32x32 RGB (3 channels)
- 1500 hidden units
- Creating a 2-layer net
- Make a cascade module
- Reshape input to vector
- Add Linear module
- Add tanh module
- Add Linear Module
- Add log softmax layer
- Create loss function module

```
Noutputs = 10;
nfeats = 3; Width = 32; height = 32
ninputs = nfeats*width*height
nhiddens = 1500

-- Simple 2-layer neural network
model = nn.Sequential()
model:add(nn.Reshape(ninputs))
model:add(nn.Linear(ninputs,nhiddens))
model:add(nn.Tanh())
model:add(nn.Linear(nhiddens,noutputs))
model:add(nn.LogSoftMax())

criterion = nn.ClassNLLCriterion()
```

Example: Training a Neural Net in Torch7

Y LeCun
MA Ranzato

```
for t = 1,trainData:size(),batchSize do
  inputs,outputs = getNextBatch()
  local feval = function(x)
    parameters:copy(x)
    gradParameters:zero()
    local f = 0
    for i = 1,#inputs do
      local output = model:forward(inputs[i])
      local err = criterion:forward(output,targets[i])
      f = f + err
      local df_do = criterion:backward(output,targets[i])
      model:backward(inputs[i], df_do)
    end
    gradParameters:div(#inputs)
    f = f/#inputs
    return f,gradParameters
  end   — of feval
  optim.sgd(feval,parameters,optimState)
end
```

- one epoch over training set
- Get next batch of samples
- Create a "closure" feval(x) that takes the parameter vector as argument and returns the loss and its gradient on the batch.
- Run model on batch
- backprop
- Normalize by size of batch
- Return loss and gradient
- call the stochastic gradient optimizer

Module-Based Deep Learning with Torch7

Y LeCun
MA Ranzato

- **Torch7 is based on the Lua language**
  - Simple and lightweight scripting language, dominant in the game industry
  - Has a native just-in-time compiler (fast!)
  - Has a simple foreign function interface to call C/C++ from Lua
- **Torch7 is an extension of Lua with**
  - A multidimensional array engine with CUDA and OpenMP backends
  - A machine learning library that implements multilayer nets, convolutional nets, unsupervised pre-training, etc
  - Various libraries for data/image manipulation and computer vision
  - A quickly growing community of users
  - Used by Facebook, DeepMind, Twitter, NYU, lots of startups
  - Supports NVIDIA's cuDNN library.
- **Torch Resource**
  - Main website: http://torch.ch
  - Cheatsheet: https://github.com/torch/torch7/wiki/Cheatsheet
  - Learn Lua in 15 minutes: http://tylerneylon.com/a/learn-lua/